MANUAL DE USUARIO

PROYECTO #1

Organización de Lenguajes y de Compiladores 1

Javier Alejandro Gutierrez de León

202004765

Guatemala, 6 de marzo de 2022

Objetivos del sistema

Objetivos Generales

✓Aplicar los conocimientos sobre la fase de análisis léxico y sintáctico de un compilador para la realización de un intérprete sencillo, con las funcionalidades principales para que sea funcional.

Objetivos Específicos

✓Reforzar los conocimientos de análisis léxico y sintáctico para la creación de un lenguaje de programación.

✓Aplicar los conceptos de compiladores para implementar el proceso de interpretación de código de alto nivel.

✓Aplicar los conceptos de compiladores para analizar un lenguaje de programación y producir las salidas esperadas.

✓Aplicar la teoría de compiladores para la creación de soluciones de software.

✓Aplicar conceptos de contenedores para generar aplicaciones livianas.

✓Generar aplicaciones utilizando arquitecturas Cliente-Servidor.

Información del Sistema

Se solicita por parte de la Escuela de Ciencias y Sistemas de la Facultad de Ingeniería, crear un lenguaje de programación para que los estudiantes de Introducción a la Programación y Computación 1, aprendan a programar y tener conocimiento de todas las generalidades de un lenguaje de programación, ya que este lenguaje será utilizado para generar sus primeras prácticas de laboratorio del curso antes mencionado. Por lo que se debe crear el proyecto llamado CompScript, el cual es un lenguaje interpretado que acpta archivos con extensión ***".cst"*** del cual se realizarán los análisis léxico, sintáctico y semántico, además de ejecutar todas las sentencias. Luego de esto se genera Reporte de Errores, en el cual se mostrarán todos los errores encontrados al realizar el análisis léxico, sintáctico y semántico. Además se generará un Árbol AST (Árbol de Análisis Sintáctico) que se debe generar una imagen del árbol de análisis sintáctico que se genera al realizar los análisis. Y por último se genra un Reporte de Tabla de Símbolos, donde se muestrán todas las variables, métodos y funciones que han sido declarados dentro del flujo del programa.

Especificación Técnica

Requisitos de Hardware

- RAM: 128 MB
- Procesador: Mínimo Pentium 2 a 266 MHz

Exploradores (Para visualizar reporte de errres):

- Google Chrome (en todas las plataformas)
- Mozilla Firefox (en todas las plataformas)
- Internet Explorer (Windows)
- Microsoft Edge (Windows, Android, iOS, Windows 10 móvil)
- Safari (Mac, iOS)
- Opera (Mac, Windows)

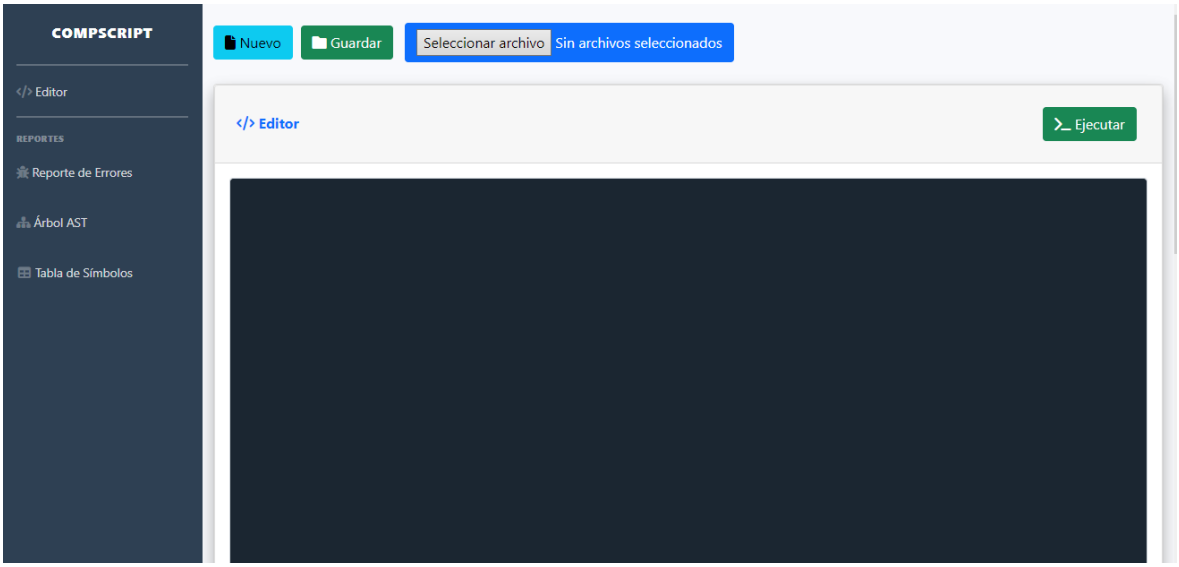
Tecnologías utilizadas

- **Node JS (16.14.0):** es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript,.
- **Angular (13.0.3):** es un framework para aplicaciones web desarrollado en TypeScript, de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página.
- **HTML5:** lenguaje de marcado para la elaboración de páginas web, con el que se realizaron los reportes de errores lexicos y sintacticos.
- **Boostrap:** Librería para el diseño del reporte HTML.
- **Graphviz:** Es un conjunto de herramientas de software para el diseño de diagramas definido en el lenguaje descriptivo DOT.
- **Javascript:** es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

Interfaz Gráfica

En esta y en las demas vistas de la interfaz tenemos un menú en el lado izquierdo en el cual podemos navegar por estas vistas:

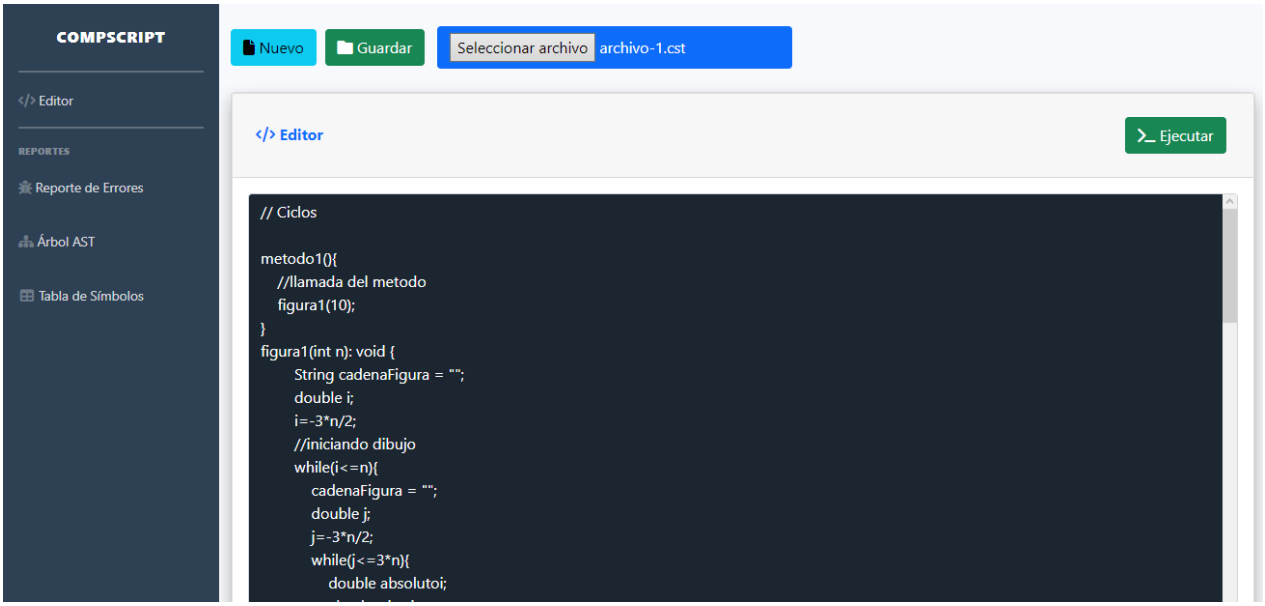
- Editor
- Reporte de errores
- Árbol AST
- Tabla de Símbolos



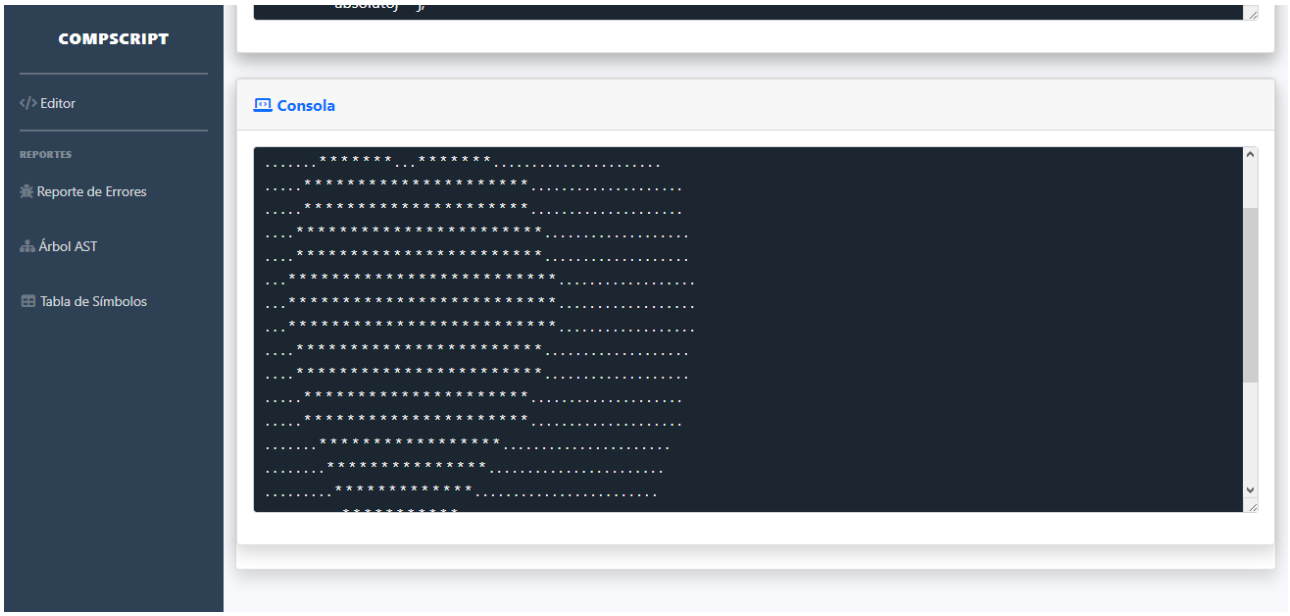
Editor

Para comenzar a utilizar el programa debemos entrar al servidor de Angular, en el cual al inicio entramos al editor donde podemos:

- Crear un nuevo archivo “cst”
- Guardar el archivo “cst”
- Abrir un archivo “cst”
- Ejecutar el código que se encuentra en el editor

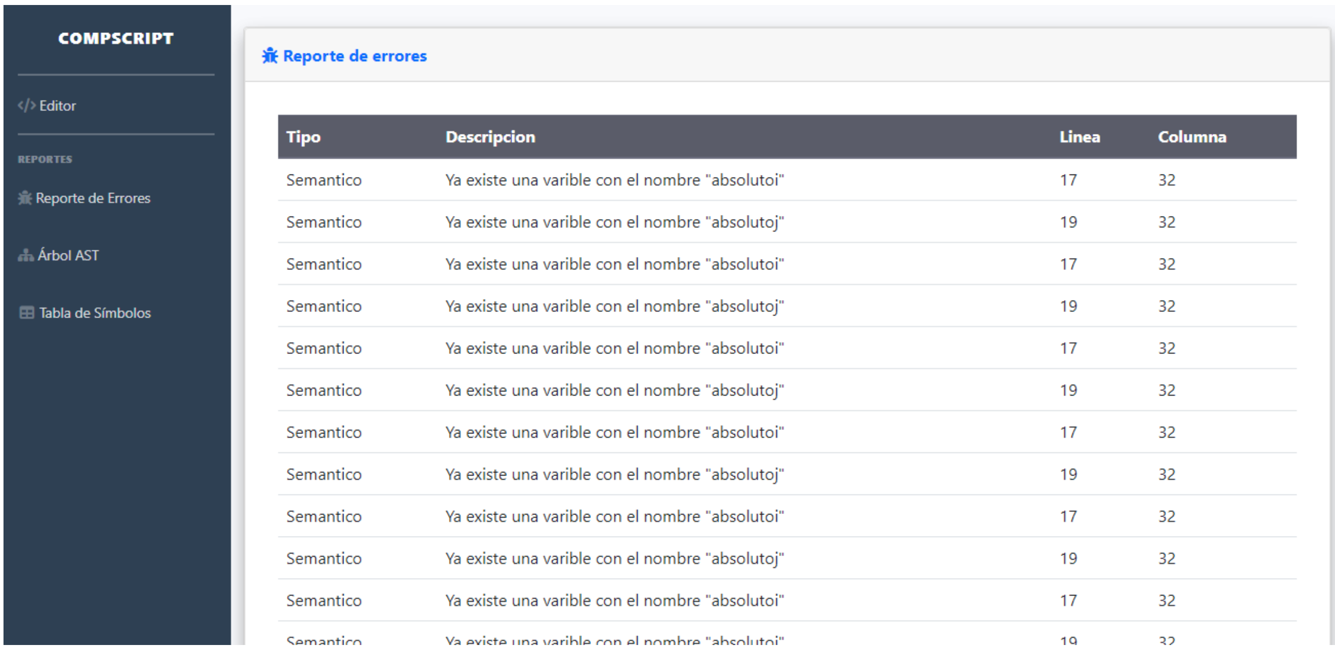


Al presionar ejecutar obtendremos el resultado de interpretar el código



Reporte de errores

En esta vista podemos observar los errores léxicos, sintácticos y semánticos que se encuentren en la ejecución, indicando fila y columna donde se encontró el error.



Arbol AST

Vista en la cual podemos visualizar el AST generado en la ejecución del programa, el cual se muestra por lo general muy grande por lo cual se puede navegar por el con scroll.

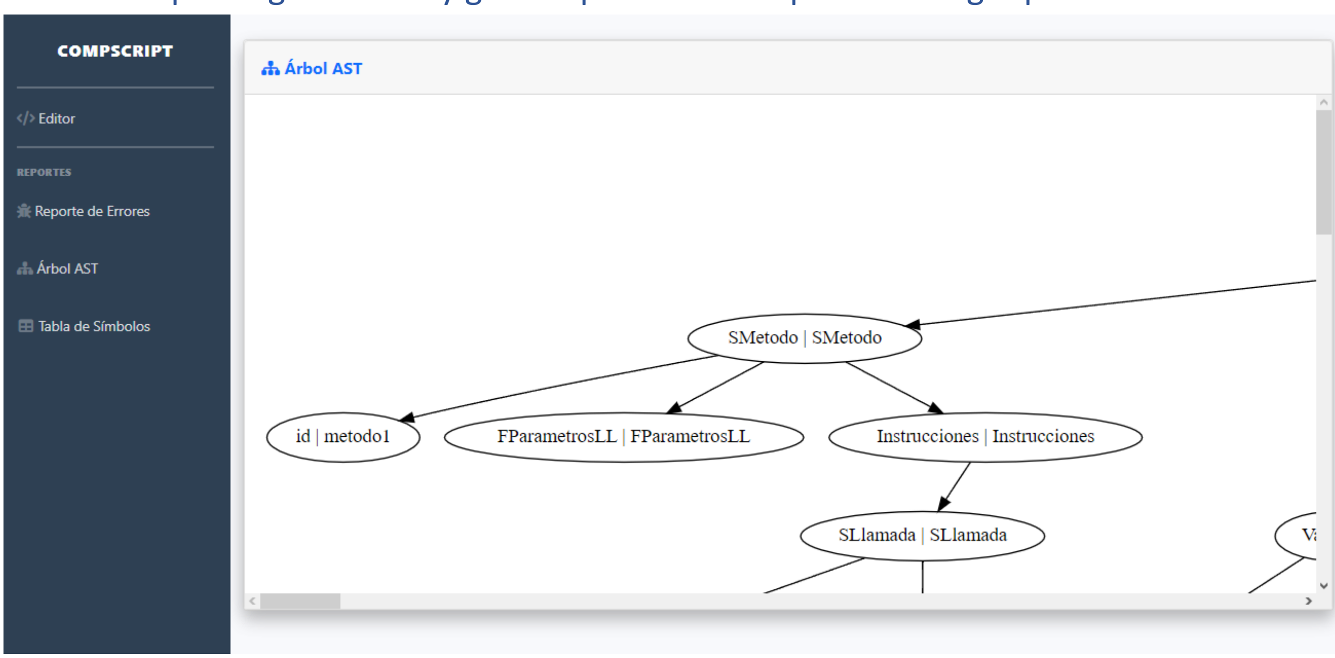
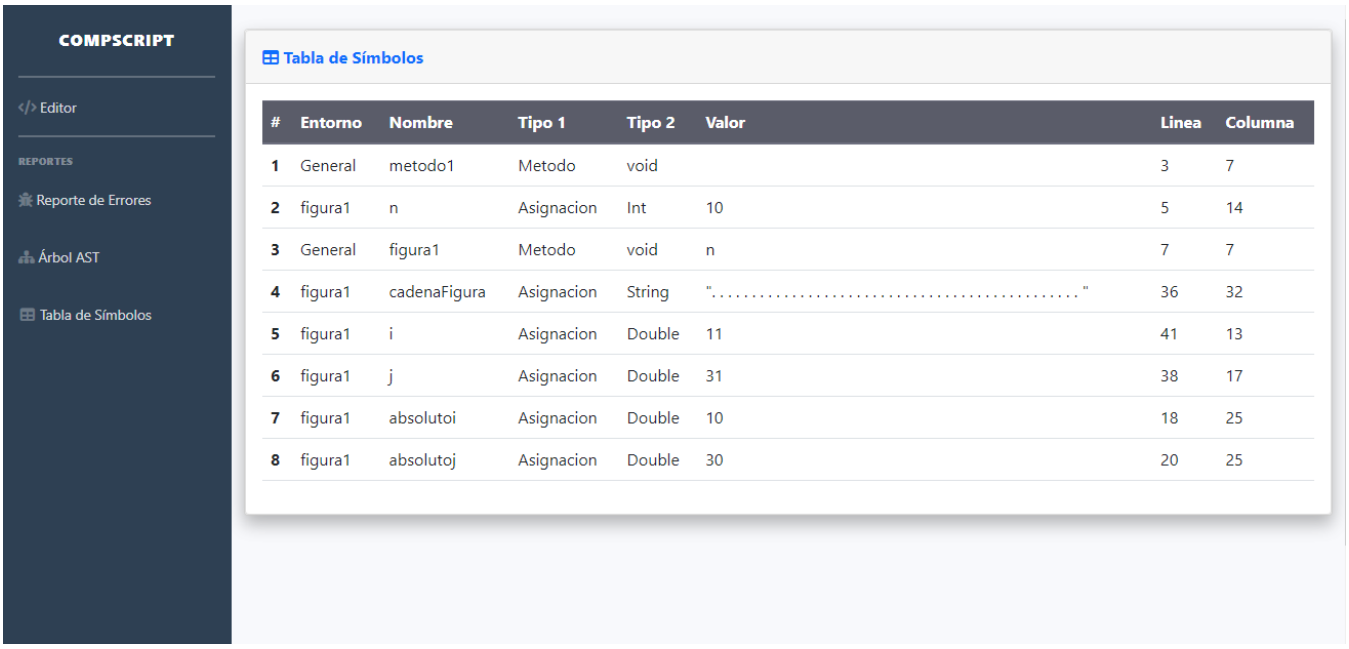


Tabla de Símbolos

En esta vista podemos visualizar la tabla de simbolos generada en la ejecución del programa.



#	Entorno	Nombre	Tipo 1	Tipo 2	Valor	Linea	Columna
1	General	metodo1	Metodo	void		3	7
2	figura1	n	Asignacion	Int	10	5	14
3	General	figura1	Metodo	void	n	7	7
4	figura1	cadenaFigura	Asignacion	String	"....."	36	32
5	figura1	i	Asignacion	Double	11	41	13
6	figura1	j	Asignacion	Double	31	38	17
7	figura1	absolutoi	Asignacion	Double	10	18	25
8	figura1	absolutoj	Asignacion	Double	30	20	25

Sintaxis del lenguaje

El lenguaje CompScript es Case-Insensitive (no distingue entre mayúsculas y minúsculas) para la sintaxis del lenguaje el cual acepta lo siguiente:

Comentarios

Comentarios de una línea

Estos comentarios deberán comenzar con // y terminar con un salto de línea.

```
// Este es un comentario de una línea
```

Comentarios multilinea

Estos comentarios deberán comenzar con /* y terminar con */.

```
/*
Este es un comentario Multilínea
Para este lenguaje
*/
```

Tipos de Datos

Los tipos de dato que soportará el lenguaje en concepto de un tipo de variable se definen a continuación:

TIPO	DEFINICION	DESCRIPCION	EJEMPLO	OBSERVACIONES	DEFAULT
Entero	Int	Este tipo de datos aceptará solamente números enteros.	1, 50, 100, 25552, etc.	Del -2147483648 al 2147483647	0
Doble	Double	Admite valores numéricos con decimales.	1.2, 50.23, 00.34, etc.	Se manejará cualquier cantidad de decimales	0.0
Booleano	Boolean	Admite valores que indican verdadero o falso.	True, false	Si se asigna un valor booleano a un entero se tomará como 1 o 0 respectivamente.	True
Caracter	Char	Tipo de dato que únicamente aceptará un único carácter, y estará delimitado por comillas simples. ' '	'a', 'b', 'c', 'E', 'Z', '1', '2', '^', '%', ' ', '=', '!', '&', '/', '\\', '\n', etc.	En el caso de querer escribir comilla simple escribir se escribirá \ y después comilla simple \', si se quiere escribir \ se escribirá dos veces \\, existirá también \n, \t, \r, \".	'\u0000' (carácter 0)
Cadena	String	Es un grupo o conjunto de caracteres que pueden tener cualquier carácter, y este se encontrará delimitado por comillas dobles. " "	"cadena1", "-- ** cadena 1"	Se permitirá cualquier carácter entre las comillas dobles, incluyendo las secuencias de escape: \" comilla doble \\ barra invertida \n salto de linea \r retorno de carro \t tabulación	"" (string vacío)

Secuencias de Escape

Las secuencias de escape se utilizan para definir ciertos caracteres especiales dentro de cadenas de texto. Las secuencias de escape disponibles son las siguientes:

SECUENCIA	DESCRIPCION	EJEMPLO
\n	Salto de línea	"Hola\nMundo"
\\	Barra invertida	"C:\\miCarpeta\\Personal"
\"	Comilla doble	"\"Esto es una cadena\""
\t	Tabulación	"\tEsto es una tabulación"
\'	Comilla Simple	"\'Estas son comillas simples\'"

Operadores Aritméticos

Suma

Es la operación aritmética que consiste en realizar la suma entre dos o más valores. El símbolo para utilizar es el signo más (+).

+	Entero	Doble	Boolean	Caracter	Cadena
Entero	Entero	Doble	Entero	Entero	Cadena
Doble	Doble	Doble	Doble	Double	Cadena
Boolean	Entero	Doble			Cadena
Caracter	Entero	Doble		Cadena	Cadena
Cadena	Cadena	Cadena	Cadena	Cadena	Cadena

Resta

Es la operación aritmética que consiste en realizar la resta entre dos o más valores. El símbolo por utilizar es el signo menos (-).

-	Entero	Doble	Boolean	Caracter	Cadena
Entero	Entero	Doble	Entero	Entero	
Doble	Doble	Doble	Doble	Doble	
Boolean	Entero	Doble			
Caracter	Entero	Doble			
Cadena					

Multiplicación

Operación aritmética que consiste en sumar un número (multiplicando) tantas veces como indica otro número (multiplicador). El sino para representar la operación es el asterisco (*).

*	Entero	Doble	Boolean	Caracter	Cadena
Entero	Entero	Doble		Entero	
Doble	Doble	Doble		Doble	
Boolean					
Caracter	Entero	Doble			
Cadena					

División

Operación aritmética que consiste en partir un todo en varias partes, al todo se le conoce como dividendo, al total de partes se le llama divisor y el resultado recibe el nombre de cociente. El operador de la división es la diagonal (/).

/	Entero	Doble	Boolean	Caracter	Cadena
Entero	Doble	Doble		Doble	
Doble	Doble	Doble		Doble	
Boolean					
Caracter	Doble	Doble			
Cadena					

Potencia

Es una operación aritmética de la forma a^b donde a es el valor de la base y b es el valor del exponente que nos indicará cuantas veces queremos multiplicar el mismo número. Por ejemplo 5^3, a=5 y b=3 tendríamos que multiplicar 3 veces 5 para obtener el resultado final; 5x5x5 que da como resultado 125. Para realizar la operación se utilizará el signo (^).

^	Entero	Doble	Boolean	Caracter	Cadena
Entero	Entero	Doble			
Doble	Doble	Doble			
Boolean					
Caracter					
Cadena					

Módulo

Es una operación aritmética que obtiene el resto de la división de un numero entre otro. El signo para utilizar es el porcentaje %.

%	Entero	Doble	Boolean	Caracter	Cadena
Entero	Doble	Doble			
Doble	Doble	Doble			
Boolean					
Caracter					
Cadena					

Negación Unaria

Es una operación que niega el valor de un número, es decir que devuelve el contrario del valor original. Se utiliza el símbolo menos (-).

-num	Resultado
Entero	Entero
Doble	Doble
Boolean	
Caracter	
Cadena	

Operadores Relacionales

Son los símbolos que tienen como finalidad comparar expresiones, dando como resultado valores booleanos. A continuación, se definen los símbolos que serán aceptados dentro del lenguaje:

Observaciones:

- Se pueden realizar operaciones relacionales entre: entero-entero, entero-doble, entero-caracter, doble-entero, doble-caracter, caracter-entero, caracter- doble, caracter-carácter y cualquier otra operación relacional entre entero, doble y carácter.
- Operaciones como cadena-carácter, es error semántico, a menos que se utilice toString en el carácter.
- Operaciones relacionales entre booleanos es válido.

OPERADOR	DESCRIPCIÓN	EJEMPLO
==	Igualación: Compara ambos valores y verifica si son iguales: - Iguales= True - No iguales= False	1 == 1 "hola" == "hola" 25.5933 == 90.8883 25.5 == 20
!=	Diferenciación: Compara ambos lados y verifica si son distintos. - Iguales= False - No iguales= True	1 != 2, var1 != var2 25.5 != 20 50 != 'F' "hola" != "hola"
<	Menor que: Compara ambos lados y verifica si el derecho es mayor que el izquierdo. - Derecho mayor= True - Izquierdo mayor= False	(5/(5+5))<(8*8) 25.5 < 20 25.5 < 20 50 < 'F'
<=	Menor o igual que: Compara ambos lados y verifica si el derecho es mayor o igual que el izquierdo. - Derecho mayor o igual= True - Izquierdo mayor= False	55+66<=44 25.5 <= 20 25.5 <= 20 50 <= 'F'
>	Mayor que: Compara ambos lados y verifica si el izquierdo es mayor que el derecho. - Derecho mayor= False - Izquierdo mayor= True	(5+5.5)>8.98 25.5 > 20 25.5 > 20 50 > 'F'
>=	Mayor o igual que: Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho. - Derecho menor o igual= True - Izquierdo menor= False	5-6>=4+6 25.5 >= 20 25.5 >= 20 50 >= 'F'

Operador Ternario

El operador ternario es un operador que hace uso de 3 operandos para simplificar la instrucción 'if' por lo que a menudo este operador se le considera como un atajo para la instrucción 'if'. El primer operando del operador ternario corresponde a la condición que debe de cumplir una expresión para que el operador retorne como valor el resultado de la expresión segundo operando del operador y en caso de no cumplir con la expresión el operador debe de retornar el valor de la expresión del tercer operando del operador.

```
<CONDICION> '?' <EXPRESION> ':' <EXPRESION>

//Ejemplo del uso del operador ternario

int edad = 18;
boolean banderaedad = false;
banderaedad = edad > 17 ? true : false;
```

Operadores Lógicos

Son los símbolos que tienen como finalidad comparar expresiones a nivel lógico (verdadero o falso). A continuación, se definen los símbolos que serán aceptados dentro del lenguaje:

OPERADOR	DESCRIPCION	EJEMPLO	OBSERVACIONES
||	OR: Compara expresiones lógicas y si al menos una es verdadera entonces devuelve verdadero en otro caso retorna falso	(55.5)|| bandera==true Devuelve true	bandera es true
&&	**AND**: Compara expresiones lógicas y si son ambas verdaderas entonces devuelve verdadero en otro caso retorna falso	(flag1) && ("hola" == "hola") Devuelve true	flag1 es true
!	NOT: Devuelve el valor inverso de una expresión lógica si esta es verdadera entonces devolverá falso, de lo contrario retorna verdadero.	!var1 Devuelve falso	var1 es true

Signos de Agrupación

Los signos de agrupación serán utilizados para agrupar operaciones aritméticas, lógicas o relacionales. Los símbolos de agrupación están dados por (y).

Precedencia de Operaciones

La precedencia de operadores nos indica la importancia en que una operación debe realizarse por encima del resto. A continuación, se define la misma.

NIVEL	OPERADOR	ASOCIATIVIDAD
0	-	Derecha
1	^	No asociativa
2	/, *	Izquierda
3	+, -	Izquierda
4	==, !=, <, <=, >, >=	Izquierda
5	!	Derecha
6	&&	Izquierda
7	||	Izquierda

Caracteres de finalización y encapsulamiento de sentencias

El lenguaje se verá restringido por dos reglas que ayudan a finalizar una instrucción y encapsular sentencias:

- **Finalización de instrucciones:** para finalizar una instrucción se utilizara el signo ;.
- **Encapsular sentencias:** para encapsular sentencias dadas por los ciclos, métodos, funciones, etc, se utilizará los signos { y }.

Declaración y asignación de variables

Una variable deberá de ser declarada antes de poder ser utilizada. Todas las variables tendrán un tipo de dato y un nombre de identificador. Las variables podrán ser declaradas global y localmente.

Durante la declaración de variables también se tendrá la opción de poder crear múltiples variables al mismo tiempo, al crear múltiples variables al mismo tiempo se tendrá la opción de crear todas las variables con un mismo valor, para ello se realizará una asignación al final del listado de las variables, en caso de no indicar esta asignación se dejará el valor por defecto para cada variable.

```
<TIPO> identificador;
<TIPO> id1, id2, id3, id4;
<TIPO> identificador = <EXPRESION>;
<TIPO> id1, id2, id3, id4 = <EXPRESION>;
//Ejemplos int numero;
int var1, var2, var3; string cadena = “hola”; char var_1 = ‘a’; boolean
verdadero;
boolean flag1, flag2, flag3 = true; char ch1, ch2, ch3 = ‘R’;
```

Casteos

Los casteos son una forma de indicar al lenguaje que convierta un tipo de dato en otro, por lo que, si queremos cambiar un valor a otro tipo, es la forma adecuada de hacerlo. Para hacer esto, se colocará la palabra reservada del tipo de dato destino ente paréntesis seguido de una expresión.

- Int a double
- Double a Int
- Int a String
- Int a Char
- Double a String
- Char a int
- Char a double

```
‘(’<TIPO>’)’ <EXPRESION>
//Ejemplos
int edad = (int) 18.6; //toma el valor entero de 18
char letra = (char) 70; //tomar el valor ‘F’ ya que el 70 en ascii es F
double numero = (double) 16; //toma el valor 16.0
```

Incremento y Decremento

Los incrementos y decrementos nos ayudan a realizar la suma o resta continua de un valor de uno en uno, es decir si incrementamos una variable, se incrementará de uno en uno, mientras que, si realizamos un decremento, hará la operación contraria.

```
<EXPRESION>’+’’+’’;’
<EXPRESION> ‘-‘‘-’’;’
//Ejemplos
int edad = 18;
edad++; //tiene el valor de 19
edad--; //tiene el valor 18

int anio=2020;
anio = 1 + anio++; //obtiene el valor de 2022
anio = anio--; //obtiene el valor de 2021
```

Estructuras de Datos

Las estructuras de datos nos sirven para almacenar cualquier valor de un solo tipo dentro de la misma estructura, en el lenguaje se tiene únicamente los vectores.

Vectores

Los vectores son una estructura de datos de tamaño fijo que pueden almacenar valores de forma limitada, y los valores que pueden almacenar son de un único tipo; int, double, boolean, char o string. El lenguaje permitirá únicamente el uso de arreglos de una o dos dimensiones.

Declaración de Vectores

Al momento de declarar un vector, tenemos dos tipos que son:

- **Declaración tipo 1:** En esta declaración, se indica por medio de una expresión numérica del tamaño que se desea el vector, además toma los valores por default para cada tipo
- **Declaración tipo 2:** En esta declaración, se indica por medio de una lista de valores separados por coma, los valores que tendrá el vector, en este caso el tamaño del vector será el de la misma cantidad de valores de la lista.

```
//DECLARACION TIPO 1 (una dimensión)
<TIPO> <ID>‘[’ ‘]’ = new <TIPO>’[’ <EXPRESION> ’]’ ‘;’
<TIPO> <ID>‘[’ ‘]’ ‘[’ ‘]’ = new <TIPO>’[’ <EXPRESION> ’]’ ’[’ <EXPRESION> ’]’
‘;’

//DECLARACION TIPO 2
<TIPO> <ID>‘[’ ‘]’ = ‘[’ <LISTAVALORES> ‘]’ ‘;’

//Ejemplo de declaración tipo 1
Int vector1[ ] = new int[4]; //se crea un vector de 4 posiciones, con 0 en
cada posición
Char vectorDosd[ ][ ] = new char [4] [4] ; // se crea un vector de dos
dimensiones de 4x4

//Ejemplo de declaración tipo 2
string vector2[ ] = ["hola", "Mundo"]; //vector de 2 posiciones, con "Hola" y
"Mundo"
char vectordosd2 [ ][ ] = [[ 0 ,0],[0 , 0]]; // vector de dos dimensiones con
valores de 0 en cada posición
```

Acceso a vectores

Para acceder al valor de una posición de un vector, se colocará el nombre del vector seguido de [EXPRESION].

```
//Ejemplo de acceso
string vector2[ ] = ["hola", "Mundo"]; //creamos un vector de 2 posiciones de
tipo string
string valorPosicion = vector2[0]; //posición 0, valorPosicion = “hola”

Char vectorDosd[ ][ ] = new char [4] [4] ; // creamos vector de 4x4
Char valor = vectorDosd[0][0]; // posición 0,0
```

Modificación de Vectores

Para modificar el valor de una posición de un vector, se debe colocar el nombre del vector seguido de '[' EXPRESION ']' = EXPRESION;

```
<ID> '[' EXPRESION ']' = EXPRESION';'
<ID> '[' EXPRESION ']' '[' EXPRESION ']' = EXPRESION';'
string vector2[ ] = ["hola", "Mundo"]; //vector de 2 posiciones, con "Hola" y
"Mundo"
int vectorNumero[ ] = [2020,2021,2022];
vector2[0] = "OLC1 ";
vector2[1] = "1er Semestre "+vectorNumero[1];
/*
RESULTADO
vector2[0]= "OLC1 "
vector2[1]= "1er Semestre 2021"
*/
```

Sentencias de control

Estas sentencias modifican el flujo del programa introduciendo condicionales. Las sentencias de control para el programa son el IF y el SWITCH.

if

La sentencia if ejecuta las instrucciones sólo si se cumple una condición. Si la condición es falsa, se omiten las sentencias dentro de la sentencia.

```
if' '(' [<EXPRESION>] ')' '{'
    [<INSTRUCCIONES>]}'
| 'if' '(' [<EXPRESION>] ')' '{'
    [<INSTRUCCIONES>]
'}' 'else' '{'
    [<INSTRUCCIONES>]
'}'
| 'if' '(' [<EXPRESION>] ')' '{'
    [<INSTRUCCIONES>]
'}' 'else' [<IF>]

//Ejemplo de cómo se implementar un ciclo if
if (x <50){
Println("Menor que 50");
}
```

Switch Case

Switch case es una estructura utilizada para agilizar la toma de decisiones múltiples, trabaja de la misma manera que lo harían sucesivos if.

```
// EJEMPLO DE SWITCH
int edad = 18;
switch( edad ) {
    Case 10: Println(“Tengo 10 anios.”);
        // mas sentencias
        Break;
    Case 18:
        Print(“Tengo 18 anios.\n”);
        // mas sentencias
    Case 25:
        Println(“Tengo 25 anios.”);
        // mas sentencias
        Break;
    Default:
    Print(“No se que edad tengo. :(\n”);
    // mas sentencias
    Break;
}
/* Salida esperada
Tengo 18 anios.
No se que edad tengo. :( */
```

Sentencias cíclicas

Los ciclos o bucles son una secuencia de instrucciones de código que se ejecutan una vez tras otra mientras la condición, que se ha asignado para que pueda ejecutarse, sea verdadera. En el lenguaje actual, se podrán realizar 3 sentencias cíclicas que se describen a continuación.

While

El ciclo o bucle While, es una sentencia que ejecuta una secuencia de instrucciones mientras la condición de ejecución se mantenga verdadera.

```
//Ejemplo de cómo se implementar un ciclo
while while (x<100){
    if (x > 50){
    Print(“Mayor que 50”);
    //Más sentencias
    }
    else{
    Print(“Menor que 100”);
    //Más sentencias
    } X++;
    //Más sentencias
}
```

For

El ciclo o bucle for, es una sentencia que nos permite ejecutar N cantidad de veces la secuencia de instrucciones que se encuentra dentro de ella.

Para la actualización de la variable del ciclo for, se puede utilizar:

- Incremento | Decremento: i++ | i—
- Asignación: como i=i+1, i=i-1, i=5, i=x, etc, es decir cualquier tipo de asignación.

```
//Ejemplo 2: asignación de variable previamente declarada y decremento por asignación
for ( i=5; i>2;i=i-1 ){
    Print(“i=”+i+”\n”)
    //más sentencias
}
```

Do-While

El ciclo o bucle Do-While, es una sentencia que ejecuta al menos una vez el conjunto de instrucciones que se encuentran dentro de ella y que se sigue ejecutando mientras la condición sea verdadera.

```
//Ejemplo de cómo se implementar un ciclo do-while Int a=5;
Do{
    If (a>=1 && a <3){ Println(true)
    }
    Else{
    Println(false)
    }
    a--;
} while (a>0);
```

Sentencias de transferencia

Las sentencias de transferencia nos permiten manipular el comportamiento de los bucles, ya sea para detenerlo o para saltarse algunas iteraciones. El lenguaje soporta las siguientes sentencias:

Break

La sentencia break hace que se salga del ciclo inmediatamente, es decir que el código que se encuentre después del break en la misma iteración no se ejecutara y este se saldrá del ciclo.

Continue

La sentencia continue puede detener la ejecución de la iteración actual y saltar a la siguiente. La sentencia continue siempre debe de estar dentro de un ciclo, de lo contrario será un error.

Return

La sentencia return finaliza la ejecución de un método o función y puede especificar un valor para ser devuelto a quien llama a la función.

Funciones

Una función es una subrutina de código que se identifica con un nombre, tipo y un conjunto de parámetros. Para este lenguaje las funciones serán declaradas definiendo primero su identificador único, luego un tipo de dato para la función, seguido de una lista de parámetros dentro de paréntesis (esta lista de parámetros puede estar vacía en el caso de que la función no utilice parámetros).

Cada parámetro debe estar compuesto por su tipo seguido de un identificador, para el caso de que sean varios parámetros se debe utilizar comas para separar cada parámetro y en el caso de que no se usen parámetros no se deberá incluir nada dentro de los paréntesis. Luego de definir la función y sus parámetros se declara el cuerpo de la función, el cual es un conjunto de instrucciones delimitadas por llaves {}.

Para las funciones es obligatorio que las mismas posean un valor de retorno que coincida con el tipo con el que se declaró la función, en caso de que no sea el mismo tipo o de que no venga un retorno dentro del cuerpo de la función debería lanzarse un error de tipo semántico.

```
<ID> ‘(‘ [<PARAMETROS>] ‘)’ ’:’ <TIPO> ‘{‘
[<INSTRUCCIONES>]
‘}’

PARAMETROS -> [<PARAMETROS] ‘,’ [<TIPO>] [<ID>]
| [<TIPO>] [<ID>]

//Ejemplo de declaración de una función de enteros
conversion(double pies, string tipo): double {
    if (tipo == “metro”){
        return pies/3.281;
    }else{
    return -1;
    }
}
```

Métodos

Un método también es una subrutina de código que se identifica con un nombre, tipo y un conjunto de parámetros, aunque a diferencia de las funciones estas subrutinas no deben de retornar un valor. Para este lenguaje los métodos serán declarados haciendo uso un identificador del método, seguido de la palabra reservada 'void' (que puede o no aparecer),seguido de una lista de parámetros dentro de paréntesis (esta lista de parámetros puede estar vacía en el caso de que la función no utilice parámetros).
Cada parámetro debe estar compuesto por su tipo seguido de un identificador, para el caso de que sean varios parámetros se debe utilizar comas para separar cada parámetro y en el caso de que no se usen parámetros no se deberá incluir nada dentro de los paréntesis. Luego de definir el método y sus parámetros se declara el cuerpo del método, el cual es un conjunto de instrucciones delimitadas por llaves {}.

```
<ID> '(' [<PARAMETROS>] ')' ':' ['void'] '{'
[<INSTRUCCIONES>]
'}'

PARAMETROS -> [<PARAMETROS] ',' [<TIPO>] [<ID>]
| [<TIPO>] [<ID>]

//Ejemplo con tipo definido de un método
holamundo(): void {
Print("Hola mundo");
}
//Ejemplo sin tipo definido de un método
HolaCompi() {
Print("Hola Compi 1");
}
```

Llamadas

La llamada a una función específica la relación entre los parámetros reales y los formales y ejecuta la función. Los parámetros se asocian normalmente por posición, aunque, opcionalmente, también se pueden asociar por nombre. Si la función tiene parámetros formales por omisión, no es necesario asociarles un parámetro real.

La llamada a una función devuelve un resultado que ha de ser recogido, bien asignándolo a una variable del tipo adecuado, bien integrándolo en una expresión.

La sintaxis de las llamadas de los métodos y funciones serán la misma.

Función Print

Esta función nos permite imprimir expresiones con valores únicamente de tipo entero, doble, booleano, cadena y carácter. Esta función no concatena un salto de línea al final del contenido que recibe como parámetro.

```
Print(“Hola mundo!!”);
Print(“Sale compi \n” + valor + “!!”);
Print(suma(2,2));
```

Función Println

Esta función nos permite imprimir expresiones con valores únicamente de tipo entero, doble, booleano, cadena y carácter. Esta función concatena un salto de línea al finalizar el contenido que recibe como parámetro.

```
Println(“Hola mundo!!”);
Println(“Sale compi \n” + valor + “!!”);
```

Función toLower

Esta función recibe como parámetro una expresión de tipo cadena y retorna una nueva cadena con todas las letras minúsculas.

Función toUpper

Esta función recibe como parámetro una expresión de tipo cadena retorna una nueva cadena con todas las letras mayúsculas.

```
string cad_1 = toLower(“hOla MunDo”); // cad_1 = “hola mundo”
string cad_2 = toLower(“RESULTADO = ” + 100); // cad_2 = “resultado = 100”
```

Round

Esta función recibe como parámetro un valor numérico. Permite redondear los números decimales según las siguientes reglas:

- Si el decimal es mayor o igual que 0.5, se aproxima al número superior
- Si el decimal es menor que 0.5, se aproxima al número inferior.

```
// Ejemplo
Double valor = round(5.8); //valor = 6
Double valor2 = round(5.4); //valor2 = 5
```

Funciones nativas

Length

Esta función recibe como parámetro un vector, una lista o una cadena y devuelve el tamaño de este.

```
//Ejemplo
string[ ] vector2 = ["hola", "Mundo"];
int tam_vector = length(vector2); // tam_vector = 2
int tam_hola = length(tam_vector[0]); // tam_hola = 4
```

Typeof

Esta función retorna una cadena con el nombre del tipo de dato evaluado.

```
Int[] lista2 =new int [];
String tipo = typeof(15); // tipo = “int”
String tipo2 = typeof(15.25); // tipo = “double”
String tipo3 = typeof(lista2); // tipo3 = “vector”
```

To String

Esta función permite convertir un valor de tipo numérico o booleano en texto.

```
String valor = toString(14); // valor = “14”
String valor2 = toString(true); // valor = “true”
```

toCharArray

Esta función permite convertir una cadena en un vector de caracteres.

```
Char[] caracteres = toCharArray(“Hola”);
/*
caracteres [0] = “H”
caracteres [1] = “o”
caracteres [2] = “l”
caracteres [3] = “a”
*/
```

Run

Para poder ejecutar todo el código generado dentro del lenguaje, se utilizará la sentencia RUN para indicar que método o función es la que iniciará con la lógica del programa.

```
funcion1():void{
    Print(“hola”);
}
run funcion1();
/*RESULTADO
hola
*/
```